User Manual

© **Maveryx** srl

All rights reserved. This document is the property of Maveryx srl. It must not be copied partly or in full without obtaining the prior written consent of Maveryx srl. Permission to copy and implement the material contained herein is granted subject to the conditions that any copy or re–publication must bear this legend in full. That any derivative work must bear a notice that it is Maveryx srl copyright document jointly published by the copyright holders and that none of the copyright holders shall have any responsibility or liability whatsoever to any other party arising from the use or publication of the material contained herein.

At the time of going to press, this paper is as thorough and correct as possible: however, information herein contained may have been updated without prior notice after this date. Maveryx srl reserves the right to change the functions of its products at any time without prior notice.

Trademarks

- Eclipse is a trademark of Eclipse Foundation Inc.
- Java and all Java–based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
- Linux is a registered trademark of Linus Torvalds in the United States and other countries.
- Windows is a registered trademark of Microsoft Corporation in the United States and other countries.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Rational is a registered trademark of International Business Machines Corporation in the United States and other countries.

All other company, product, or service names contained on this site may be trademarks or service marks of others and are the property of their respective owners.

Welcome to Maveryx 5

About this guide 5

Definitions, Acronyms and Abbreviations 6

1. About Maveryx 7

1.1 Avoiding GUI maps 7
1.2 GUI Objects Finder 7
1.3 Extensibility 8
1.3.1. Automating custom objects 9
1.3.2. Adding matching algorithms 9
1.3.3. Maveryx APIs add-on 9
1.3.4. Maveryx Eclipse plug-in 9

2. Installing Maveryx 10

2.1 System Requirements 10
2.1.1. Java Version 10
2.1.2. Operating System 10
2.1.3. Eclipse Version 10
2.2 Getting Maveryx 10
2.3 Installing Maveryx 11
2.3.1. Installation on Windows 11
2.3.2. Installation on Linux/Unix/Mac 11
2.3.2.1 Maveryx Framework 11
2.3.2.2 Eclipse Plug-in 12

3. Maveryx plug-in for Eclipse 13

3.1 Configuring the Maveryx Plugin 13
3.2 Switching into the Maveryx Perspective 13
3.3 Creating a new Maveryx project 15
3.4 Creating a new Maveryx Test Class 17
3.5 Building the project 19
3.6 Running the project 20
3.7 Viewing Test Results in Eclipse 21

4. Getting Started With Maveryx 23

4.1 Running the examples 23
4.1.1. Creating a Project for the Maveryx Demo Code 23
4.1.2. Running the data-driven testing example 24
4.1.3. Running the Java test class example 25
4.1.4. Running the JUnit Test examples 25
4.2 Writing an automated test 26
4.2.1. Creating a new Test Script using JUnit 26

5. Testing with Maveryx 30

5.1 Creating a new test project 30
5.2 Configuring the launch file 32
5.2.1. Using the Application Configuration tool 32
5.2.2. Setting up the launch file 33
5.3 Configuring the GUI Object Finder 35
5.3.1. Configure the Searching Criteria 35
5.3.1.1 Configuring the LevenshteinDistance 39
5.3.1.2 Configuring the SynonymousMatch 39
5.3.1.3 Configuring the GoogleTranslation 39
5.3.2. Configuring the Evaluation Function 40
5.4 Tricks and Tips 41
5.4.1. Launching the application under test 41
5.4.2. Creating test objects 41

5.4.2.1 Creating multiple objects 42
5.4.3. Getting info on test objects 43
5.4.4. waitForObject() 43
5.4.5. Logging 44
5.4.6. Reporting 44
5.4.7. Data–driven testing 45
5.4.7.1 Getting test data from an XML file 45
5.4.7.2 Getting test data from an Excel file 47
5.4.8. Verification Points 48
5.4.9. Distributed and Multi-Application Testing 48

Welcome to Maveryx

Maveryx is a free and open source test automation tool for functional and regression testing of Java applications. It provides testers with automated testing capabilities for functional testing, regression testing, GUI testing and data–driven testing.

Maveryx completely eliminates the "GUI Map" (or "Test Object Map") dependency. You don't have to capture and maintain any GUI map: the user interface is investigated directly at runtime and the test objects are identified through an *Intelligent GUI Objects Finder* powered by a wide set of searching algorithms. Maveryx uses approximate (or fuzzy) matching algorithms to locate objects even in case of partial information given in the test scripts or if the test objects have changed since test creation. Maveryx significantly reduces the time and effort testers spend on creating and maintaining test scripts and enables earlier testing.

Maveryx is provided as stand-alone application and Eclipse plug-in. Maveryx plug-in for Eclipse aims to provide all tool functionality to the Eclipse development environment. The plug-in allows you to edit your test script in Eclipse, making it easier and faster to create and run your automated tests.

ABOUT THIS GUIDE

This guide describes how to use Maveryx to test your applications. It provides step–by–step instructions to help you create, debug and run tests.

- Chapter 1 introduces Maveryx
- Chapter 2 describes how to get and install Maveryx
- Chapter3 contains a tutorial on using the Eclipse Plug-in for Maveryx
- Chapter 4 helps you get started with Maveryx
- Chapter 5 walks through the different functionalities of Maveryx

DEFINITIONS, ACRONYMS AND ABBREVIATIONS

Def.	Description
Approximate Matching	Approximate string matching is the technique of finding approximate matches to a pattern in a string.
Assertion	An assertion is a statement describing a specific set of conditions expected to be met.
Data–Driven Testing	A methodology used in test automation where test scripts are executed and verified based on the data values stored in one or more central data sources or databases.
GUI map	A GUI or test object map or object map is a static view that lists the test objects in the application–under–test. It contains the recognition properties for each test object in the application being tested.
Levenshtein distance	The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other.
Verification point	A checkpoint verifies that a certain action has taken place, or verifies the state of an object.

1. About Maveryx

Maveryx is a professional automated functional and regression testing tool that enables you to test Java applications. With Maveryx you can quickly create and edit simple, easy–to–read, Java test scripts, and you can test any object in the application–under–test, including the object's properties and data.

Maveryx provides everything you need to quickly create and run tests.

1.1 Avoiding GUI maps

One of the major challenges in test automation is to deal with Graphical User Interface (GUI) controls. Many automated testing tools use the so–called 'GUI maps' to recognize and locate objects in the application being tested. A GUI map is a static view that describes the test objects in the application–under–test. The automation tool reads an object's description in the GUI map and then looks for an object with the same properties in the application being tested.

The object maps for an application are created when recording test cases or by using a spy tool when writing script code (it is quite infeasible to create a map manually from scratch!). Such maps are mandatory to create, build and run the test scripts, so it is impossible to develop the automated tests before the availability of the application to test.

Moreover, when the user interface is modified, GUI maps, which embody assumptions about how to identify the interface's objects, need to be updated to reflect changes.

Differently from other tools, Maveryx analyzes and identifies test objects and their properties and values dynamically at runtime during the scripts execution, without using any object map.

Running a test, Maveryx, like a photographer, gets 'snapshots' of the current application's user interface. Each snapshot is processed by an *Intelligent GUI Objects Finder* that recognizes and locates the test objects by using some advanced, fast and efficient, searching algorithms. Maveryx analyzes and identify each object in the application–under–test much the same way that a person looks at a photograph and points out one item in the picture. Maveryx uses a very human–like technique for identifying objects during the test run.

Avoiding GUI maps you can develop the automated test scripts early in the software lifecycle, long before the application is released for testing.

To ease the maintenance of the test scripts when the application–under–test changes, you can use a data–driven approach (5.4.7) to create a GUI abstraction layer (objects repository) in order to separate tests from the user interface data, without having to make further changes to the scripts.

1.2 GUI Objects Finder

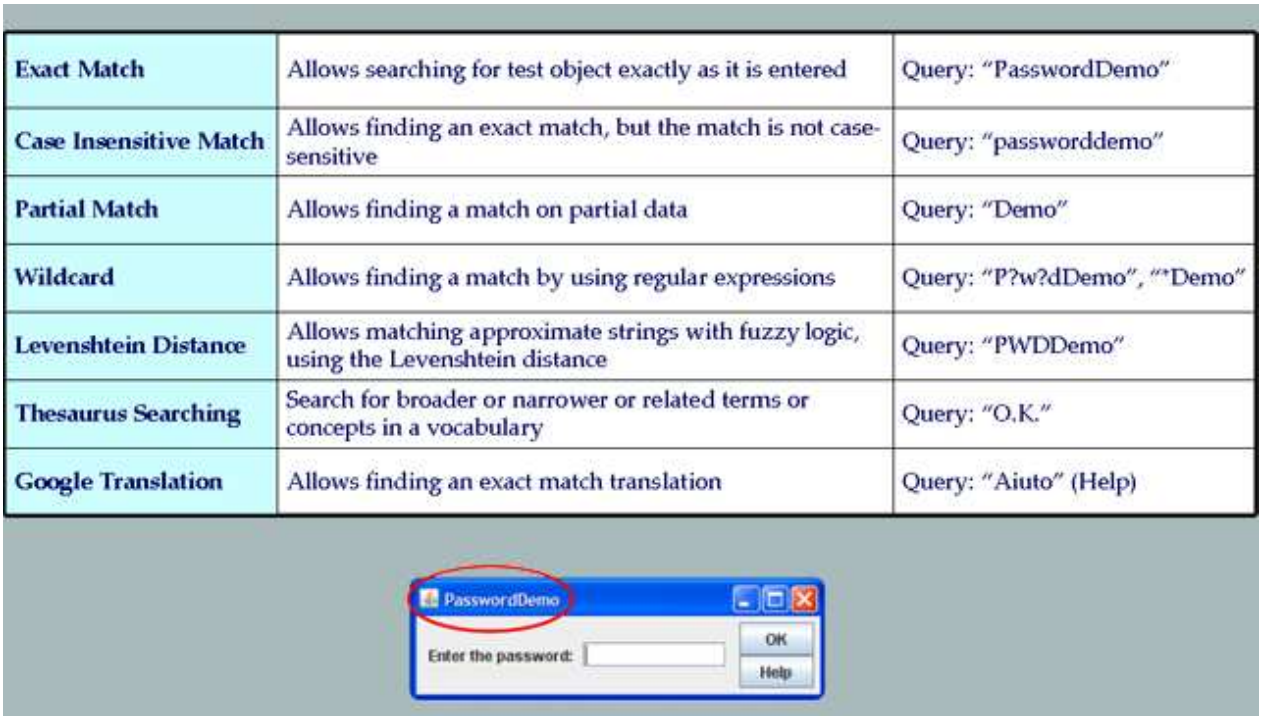
Test objects defined in the scripts are identified and located directly at runtime by an advanced *GUI Objects Finder* with several pattern–matching capabilities (5.3). This technology significantly reduces the time and effort testers spend on script maintenance and enables earlier testing.

Getting an early start on test automation puts you on the road to success. But writing test scripts early from specifications is not always "reliable", because requirements or design documents in most cases are unstable, contains partial information or are lacking. Incomplete information is inevitably propagated from specifications to test scripts, causing failure during execution.

Moreover, changing the user interface, for example adding something, will often break the test cases which depend on the modified parts. Typically, even minor changes to the GUI layout break the test scripts. GUI modifications make a large number of test cases unusable, requiring expensive update.

Maveryx uses advanced matching algorithms to search and locate objects during the test run, even if the objects have changed since test creation or 'partial' descriptions are given. Maveryx searches for run–time objects that exactly or partially match the test objects in the test scripts. Advanced matching algorithms (**Figure 1**), including fuzzy ones, allow uniquely identifying test objects by approximate matches.

You can configure the object–matching sensitivity to run the test scripts successfully, without changing the scripts, even when the application–under–test has been updated. This feature not only avoids script maintenance, but also enables testers to test the functionality of the application before the UI design is finalized.



Exact Match	Allows searching for test object exactly as it is entered	Query: "PasswordDemo"
Case Insensitive Match	Allows finding an exact match, but the match is not case-sensitive	Query: "passworddemo"
Partial Match	Allows finding a match on partial data	Query: "Demo"
Wildcard	Allows finding a match by using regular expressions	Query: "P?w?dDemo", "*Demo"
Levenshtein Distance	Allows matching approximate strings with fuzzy logic, using the Levenshtein distance	Query: "PWDDemo"
Thesaurus Searching	Search for broader or narrower or related terms or concepts in a vocabulary	Query: "O.K."
Google Translation	Allows finding an exact match translation	Query: "Aiuto" (Help)

Figure 1 – GUI Objects Finder matching algorithms

1.3 Extensibility

Maveryx's plug-in system makes it easy to customize and extend Maveryx for your specific testing needs.

1.3.1. Automating custom objects

Maveryx may not recognize customized user interface objects. You can extend Maveryx to support new custom GUI components as well as to support new platforms.

For further info, please contact us at support@maveryx.com.

1.3.2. Adding matching algorithms

The GUI Objects Finder can be extended with new matching functionality best fitting the test objects in your application domains.

For further info, please contact us at support@maveryx.com.

1.3.3. Maveryx APIs add-on

Maveryx provide a comprehensive set of high-level APIs for simple development of powerful test scripts. You can create your own API to extend the platform with new powerful testing capabilities.

For further info, please contact us at support@maveryx.com.

1.3.4. Maveryx Eclipse plug-in

Maveryx is available as plug-in for Eclipse. It is designed to give you a powerful, integrated environment in which to build and run your Maveryx tests. Maveryx plug-in extends the capabilities of Eclipse to let you quickly set up new Maveryx projects, create test classes based on the Maveryx Framework API, compile, run, and debug Maveryx scripts and classes from the Eclipse SDK. Developing in Eclipse with Maveryx is highly recommended and is the fastest way to get started.

2. Installing Maveryx

2.1 System Requirements

Maveryx requires your computing environment meets some minimum requirements.

2.1.1. Java Version

Maveryx requires a fully compliant J2SE Java Runtime Environment 1.6 or later[1].

2.1.2. Operating System

Maveryx is a 100% Java application and should run correctly on any system that has a compliant Java implementation.

Maveryx has been tested and works with:

- Windows XP, Windows Vista, Windows 7
- Unix (Linux)
- Mac OS X

2.1.3. Eclipse Version

To install the Maveryx plug-in for Eclipse you must have the following:

- Eclipse Ganymede or later (version 3.4)[2]

2.2 Getting Maveryx

The easiest way to begin using Maveryx is to first download the latest production release and install it.

The latest stable version of Maveryx is available at http://sourceforge.net/projects/maveryx/.

[1] http://java.sun.com/javase/downloads/index.jsp

[2] http://eclipse.org/downloads

2.3 Installing Maveryx

2.3.1. Installation on Windows

Windows users can download an installer for the relevant version of Maveryx. To install Maveryx run the Setup program following the on-screen instructions. The setup installs both the Maveryx Framework and the Eclipse plug-in.

NOTE: If you are upgrading your installation of Maveryx from a previous version, you will need to uninstall the old Maveryx version before installing the new version.

2.3.2. Installation on Linux/Unix/Mac

2.3.2.1 Maveryx Framework

To install a new release of Maveryx you will need to download the relevant pack .zip archive, once downloaded find a suitable installation location on your machine and extract the zipped files (e.g. /Maveryx).

The installation directory structure should look something like this (**Figure 2**):

MAVERYX_HOME

- ⇒ /bin //contains the executable JAR files
- ⇒ /bin/log //the directory where the log file will be generated
- ⇒ /bin/report //the directory where the report file will be generated
- ⇒ /bin/remote //the directory for remote testing
- ⇒ /demo //contains samples (including source code)
- ⇒ /doc //contains documentation
- ⇒ /lib //contains the library (dependency) files
- ⇒ /temp //contains temporary files

Figure 2 – Maveryx installation directory

2.3.2.2 Eclipse Plug-in

To install the Maveryx Eclipse plug-in you have to copy the jar files containing the plug-in into the ***plugins*** folder of your Eclipse installation directory. For example, if Eclipse is installed in c:\eclipse, copy the jar file to c:\eclipse\plugins.

3. Maveryx plug-in for Eclipse

This guide contains instructions on how to get the most out of the Maveryx plug-in for Eclipse. Please make sure that you have followed the installation instructions first and have successfully installed the plug-in before continuing.

3.1 Configuring the Maveryx Plugin

After you've successfully downloaded and installed the Maveryx plugin as described above, the next step is to modify your Maveryx preferences in Eclipse to point to the Maveryx directory:

1. Select **Window** → **Preferences** to open the Preferences panel (**Figure 3**)
2. Select ***Maveryx*** from the left panel
3. For the *Maveryx Location* in the main panel, click **Browse...** and locate your Maveryx directory
4. Click **Apply**, and then **OK**.

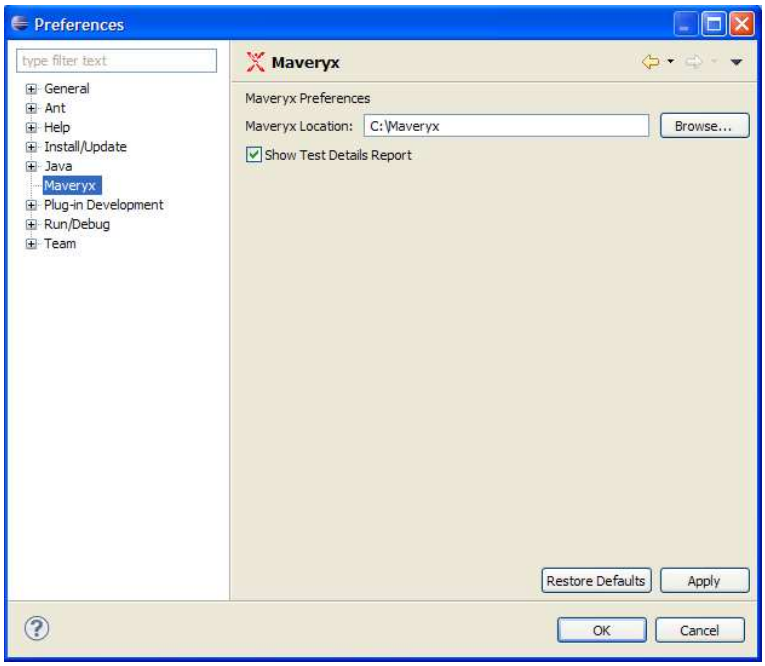


Figure 3 – Maveryx Preferences

3.2 Switching into the Maveryx Perspective

Perspectives in Eclipse are opened using the ***Window->Open Perspective*** menu. The initial menu shows the more common perspectives that are available.

To open the Maveryx perspective select **Window** → **Open Perspective** → **Other...** (**Figure 4**).

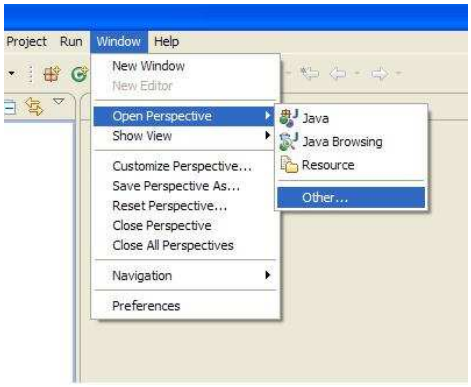


Figure 4 – Open Perspective menu

In the *Open Perspective* window select **Maveryx** and click **OK** (**Figure 5**).

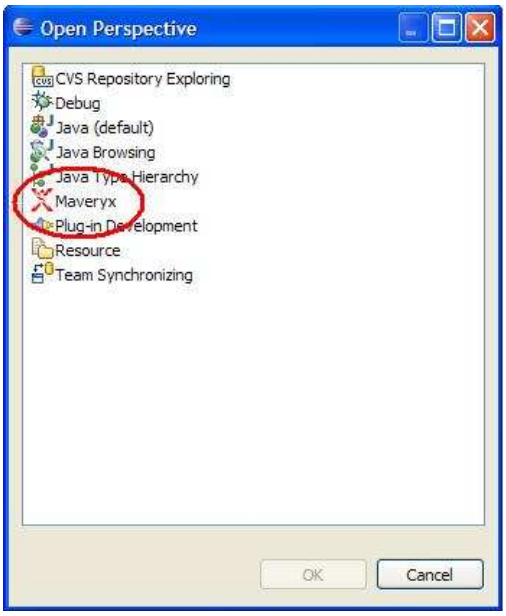


Figure 5 – Maveryx Perspective

After you have opened the perspective, Maveryx is displayed in the title bar and on the perspective shortcut bar on the left-hand side of your Eclipse window (**Figure 6**). In the future, you can switch into the Maveryx perspective by clicking on this icon, without having to use the Window-> Open Perspective menu.

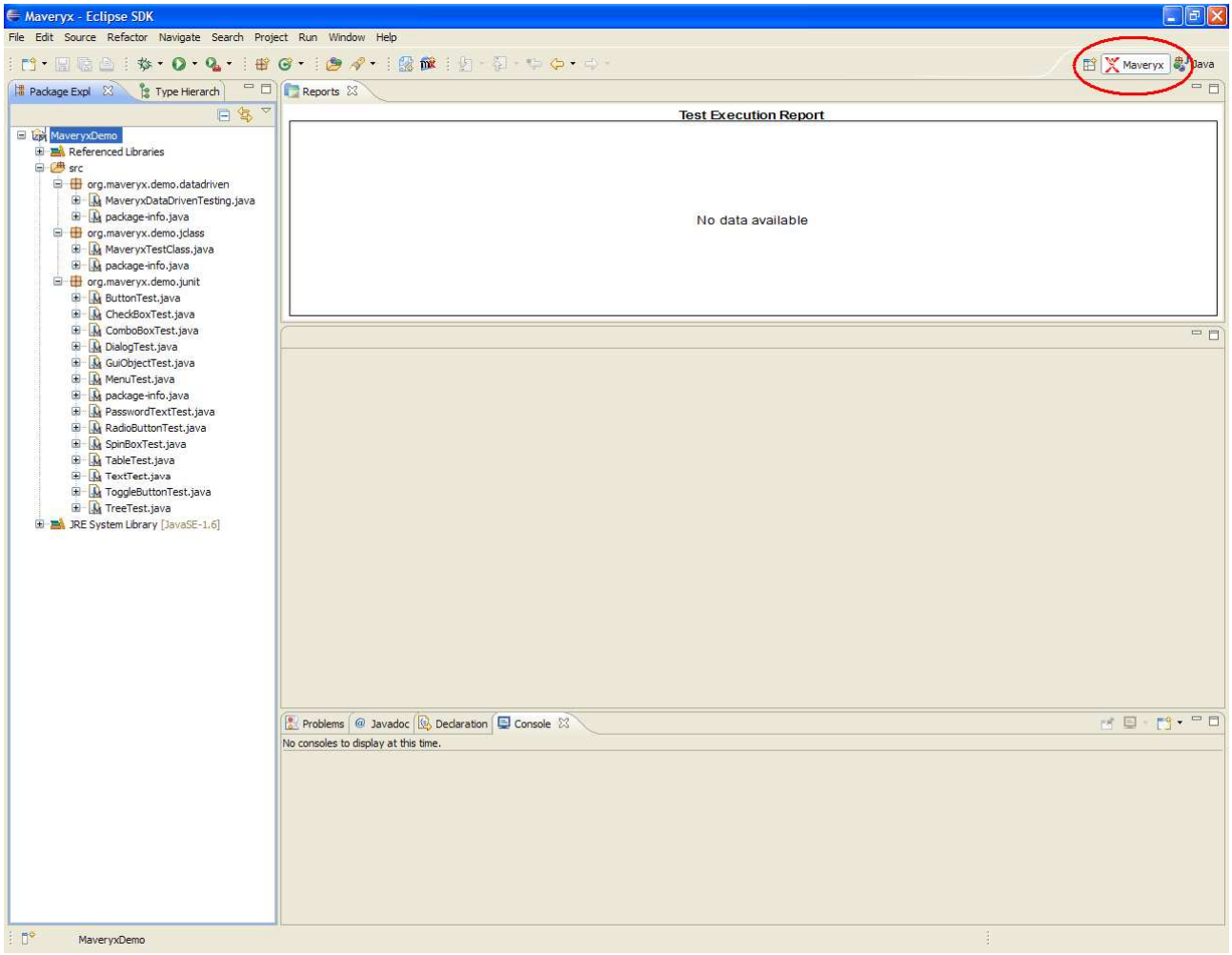


Figure 6 – Maveryx Perspective button

3.3 Creating a new Maveryx project

A Maveryx project contains source code and related files for building a test suite. It has an associated Java builder that can incrementally compile Java source files as they are changed.

To create a new Maveryx test project in workspace:

1. Launch the Eclipse IDE
2. In the Maveryx perspective (3.2):
    a. select **File** → **New** → **Maveryx Test Project** (**Figure 7**) or
    b. click the [ ] button on the toolbar
3. In the ***Maveryx Test Project*** window (**Figure 8**)
    a. enter a name for the project
    b. in the ***JRE*** section make sure that Java/JRE 6 or higher is selected
4. Click **Finish** to create the test project

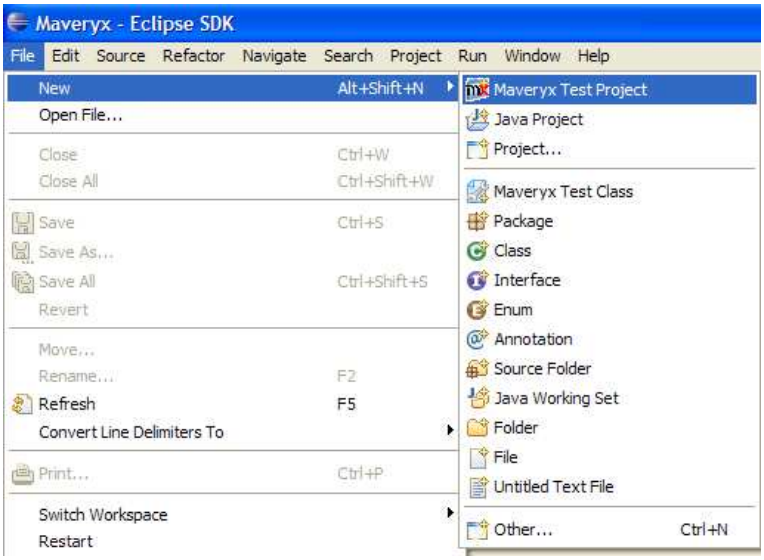


Figure 7 – File → New → Maveryx Test Project

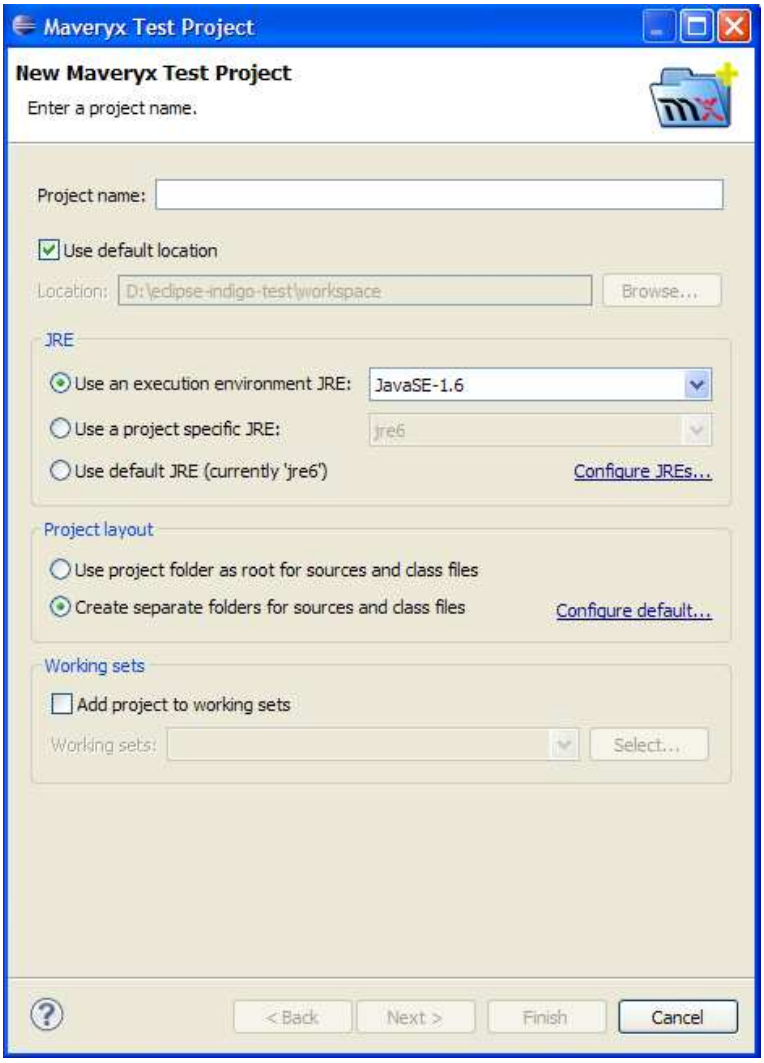


Figure 8 – New Maveryx Test Project

3.4 Creating a new Maveryx Test Class

In this section, you will create a new Maveryx test class and add methods using code generation actions.

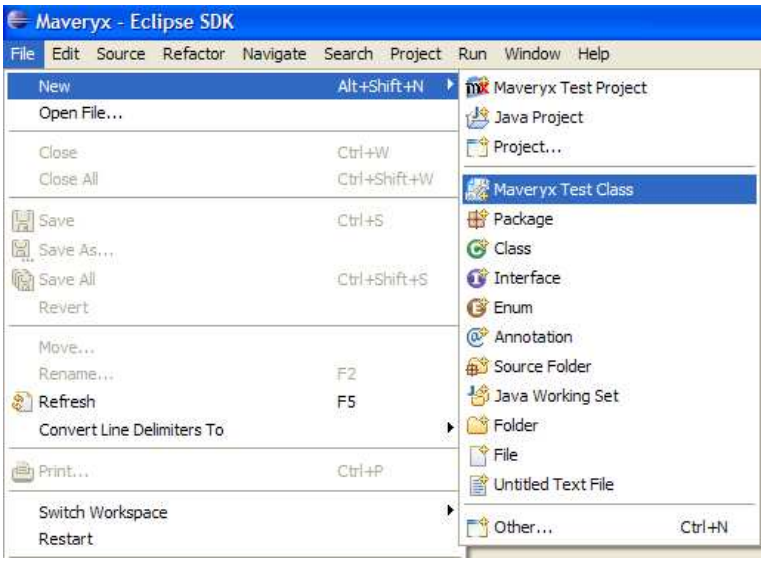


Figure 9 – File → New → Maveryx Test Class

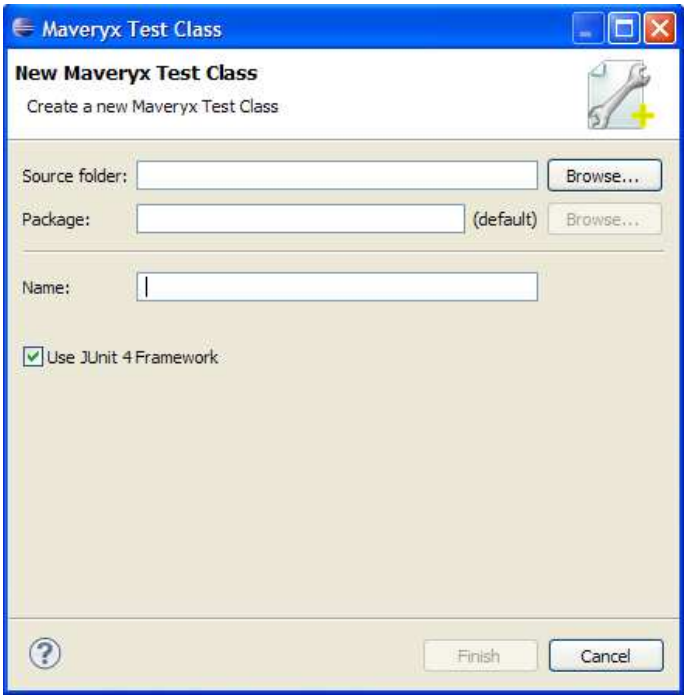


Figure 10 – New Maveryx Test Project

To create a new Maveryx test class:

1. Launch the Eclipse IDE
2. In the Package Explorer view, select the Maveryx project
3. In the Maveryx perspective (3.2),
    a. select **File** → **New** → **Maveryx Test Class** (**Figure 9**) or

b. click the [ ] button on the toolbar

4. In the ***Maveryx Class*** window enter (**Figure 10**)
    a. a name for the package
    b. a name for the class
5. Click ***Use JUnit 4 Framework*** to create your test class as JUnit 4 Test (recommended)
6. Click **Finish** to create the test class
7. The new Java file is opened in the editor. It contains the new class, the constructor, the setUp() and tearDown() methods, a test case stub (*test001()*) and many comments

```
package org.maveryx.demo;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.maveryx.bootstrap.Bootstrap;

@RunWith(org.maveryx.test.junit.MaveryxTestRunner.class)
public class MyTest {

    /**
     * Change this path to your current application's XML launch file.
     */
    private final String pathName = "C:\\Maveryx\\demo\\ AUTconfiguration.xml ";

    /**
     * Default constructor.
     * @throws Exception
     */
    public MyTest() throws Exception {
        super();
    }

    /**
     * Start the Application-Under-Test.
     * @throws Exception
     */
    @Before
    public void setUp() throws Exception {
        Bootstrap.startApplication(pathName); //start the application under test
    }

    /**
     * Close the Application-Under-Test.
     * @throws Exception
     */
    @After
    public void tearDown() throws Exception {
        Bootstrap.stop(pathName); //close the application under test
    }

    /**
     *  Test 1
```

```
     * @throws Exception
     */
    @Test
    public void test001() throws Exception {
        //Write here your test case
    }
}
```

If you don't use JUnit 4 a new Java class is opened in the editor. It contains the new class, the *startApp()* method to launch the application under test, the *main* method and many comments

```
package org.maveryx.demo;

import org.maveryx.bootstrap.Bootstrap;

public class MyTestClass {

    /**
     * Change this path to your current application's XML launch file.
     */
    private static final String pathName = "C:\\Maveryx\\demo\\ AUTconfiguration.xml ";

    /**
     * Start the Application-Under-Test by loading the launch parameters from an XML file.
     * @param xmlFile - pathname string of the XML file containing the launch parameters
     * @throws Exception
     */
    private static void startApp(String xmlFile) throws Exception {
        Bootstrap.startApplication(pathName); //start the application under test
    }

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        startApp(pathName); //launch the application under test

        //write here your test case

        Bootstrap.stop(pathName); //close the application under test

        Bootstrap.stop(); //close the JVM running the application under test

    }

}
```

3.5 Building the project

Once you have added code to your test project, to build the project:

1. In the Package Explorer view, select the Maveryx project

2. Select **Project** → **Build Project** from the Eclipse main menu. Alternatively, you can right-click the project and select Build Project from the context menu.

Note:

By default Eclipse is configured to perform builds automatically when you save files. To disable this feature and only perform manual builds, ensure that **Project** → **Build Automatically** from the Eclipse main menu is not checked.

3.6 Running the project

To run a test script as JUnit Test (**Figure 11**):

1. In the Package Explorer, select the Java compilation unit containing the test you want to launch
2. Press the Run [ ] button in the workbench toolbar or select **Run** → **Run** from the workbench menu. Alternatively, select **Run As** → **JUnit Test** from the Package Explorer pop-up menu, select **Run** → **Run As** → **JUnit Test** in the workbench menu bar, or select **Run As** → **JUnit Test** in the drop-down menu on the Run tool bar button
3. Your test is now launched

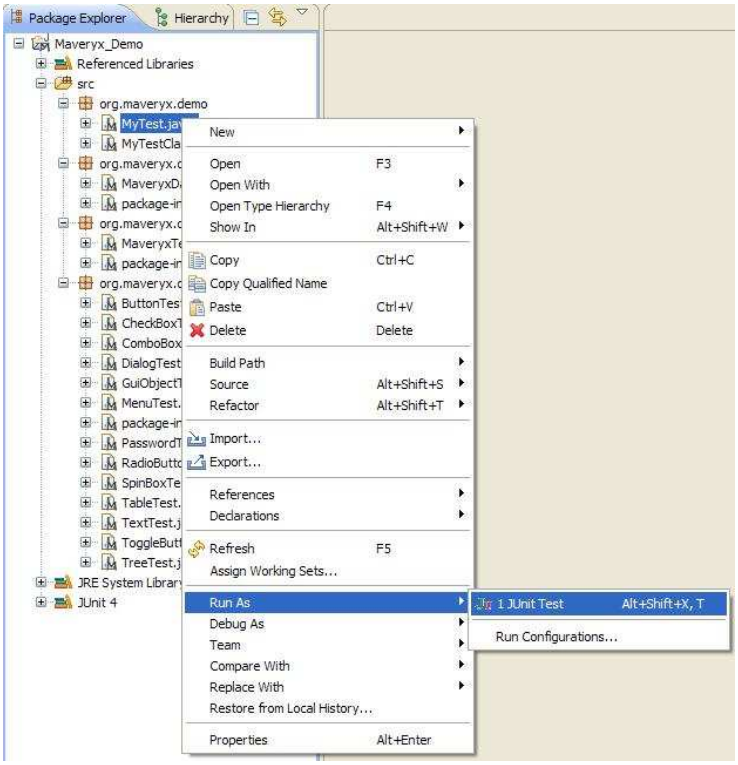


Figure 11 – Run as JUnit Test

To run a test script as Java class:

1. In the Package Explorer, select the Java compilation unit containing the test you want to launch

2. Press the Run [ ] button in the workbench toolbar or select **Run** → **Run** from the workbench menu. Alternatively, select **Run As** → **Java Application** from the Package Explorer pop-up menu, select **Run** → **Run As** → **Java Application** in the workbench menu bar, or select **Run As** → **Java Application** in the drop-down menu on the Run tool bar button
3. Your test is now launched

You can also launch your test scripts by selecting the Maveryx project instead of the compilation unit

3.7 Viewing Test Results in Eclipse

Maveryx comes with powerful reports and metrics to enable users to quickly and easily interpret the test results.

Before executing the tests, to open the ***Report*** view[3], in the Eclipse toolbar, select **Window** → **Show View** → **Reports** (**Figure 12**)

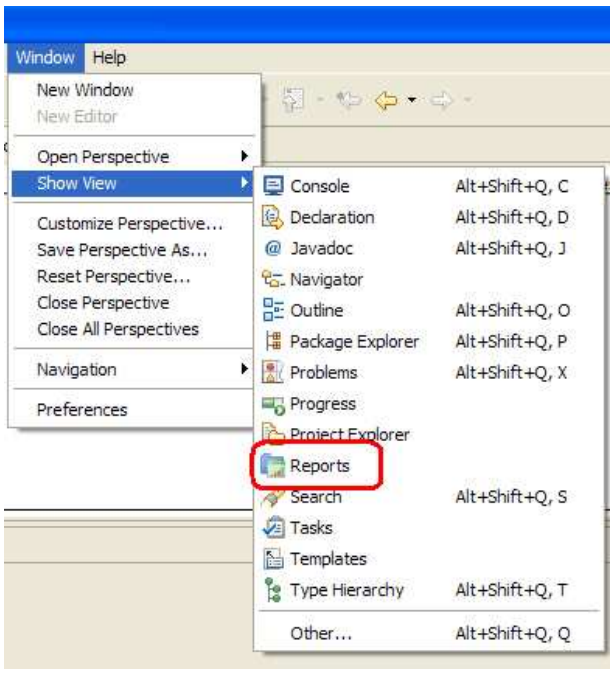


Figure 12 – Open the Report view

[3] The *Report View* is automatically loaded when the *Maveryx Perspective* is opened (3.2).

When you run your tests, Maveryx displays the status of the tests (**Figure 13**) in the ***Report*** view.

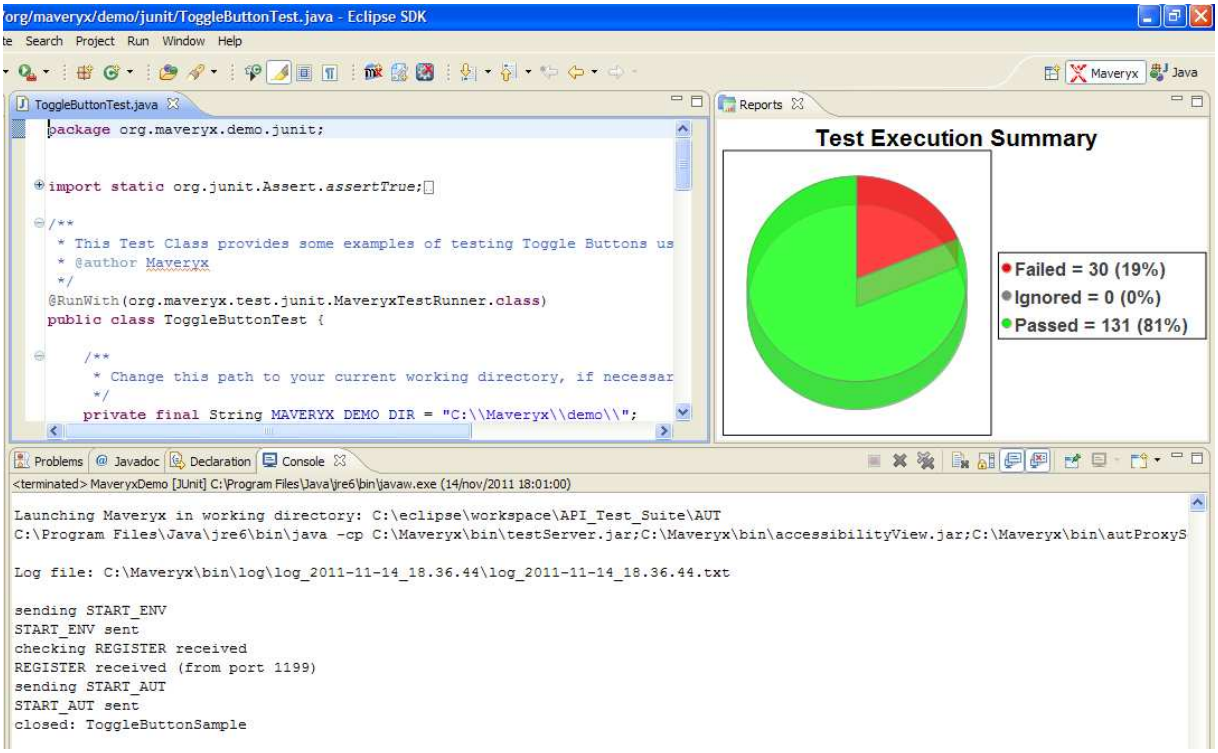


Figure 13 – Test Execution Status

To view more detailed reports and metrics at the end of each test execution:

1. Select **Window** → **Preferences** to open the Preferences panel (**Figure 3**)
2. Select ***Maveryx*** from the left panel
3. Select ***Show Detailed Test Reports***
4. Click **Apply**, and then **OK**.

After executing all tests the Test Execution Details window (**Figure 14**) will appear listing the status of all tests.

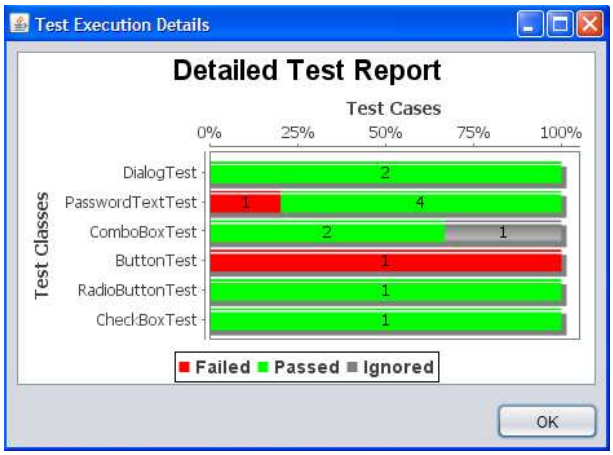


Figure 14 – Test Execution Details

4. Getting Started With Maveryx

4.1 Running the examples

The section contains the instructions to get you started using Maveryx through the examples provided in MAVERYX_HOME/demo.

Although Maveryx is IDE–independent[4], in the following sections we will use Eclipse as Java Development Environment and the Maveryx plug-in.

4.1.1. Creating a Project for the Maveryx Demo Code

Maveryx ships with a set of tutorials and demo tests. You have to create a project in Eclipse for the demo code.

To create a project for demo code:

1. Launch the Eclipse IDE
2. In the Maveryx perspective (3.2), select **File** → **New** → **Other …**
3. In the ***New*** window select **Maveryx** → **Maveryx Demo** (**Figure 15**)
4. Click **Next** to continue
5. In the ***Maveryx Demo Project*** window
    a. enter a name for the project (e.g. "MaveryxDemo")
    b. in the ***JRE*** section make sure that Java/JRE 6 or higher is selected
6. Click **Finish** to create the new project

The demo project consists of three packages:

1. *org.maveryx.demo.datadriven*: this package contains an example of data-driven testing
2. *org.maveryx.demo.jclass*: this package contains an example of test script implemented as Java class
3. *org.maveryx.demo.junit*: this package contains several test scripts implemented as JUnit tests. Each script provides a brief example for testing a kind of GUI object (e.g. push button, check box, dialog …).

[4] You can use Maveryx with all Java IDEs such as Eclipse, NetBeans, IntelliJ IDEA, JEdit etc. You can also use Maveryx with other Java-based test automation tools (e.g. JUnit, IBM Rational Functional Tester …).

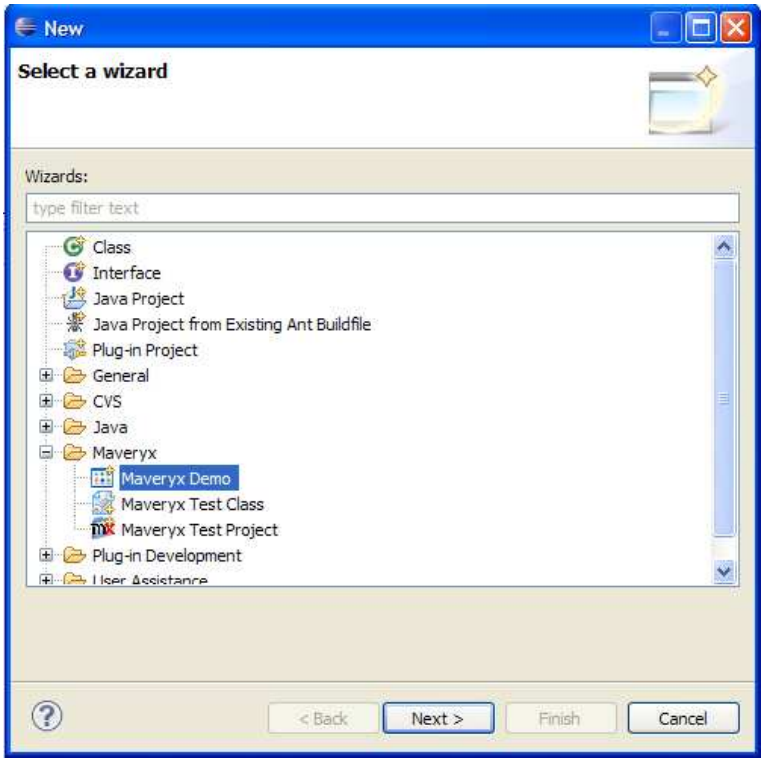


Figure 15 – New Maveryx demo project

4.1.2. Running the data-driven testing example

1. Create a new project as in 4.1.1
2. Select the project's folder **src**
3. Open the resource *org.maveryx.demo.datadriven → MaveryxDataDrivenTesting.java*
4. Change[5] the value of the string *MAVERYX_DEMO_DIR* to the Maveryx "demo" directory (MAVERYX_HOME/demo)
5. Open file *LoginDemo.xml* and change[6]:
    a. <WORKING_DIR> C:\Maveryx\demo\AUT </WORKING_DIR> to the MAVERYX_HOME/demo/AUT directory
    b. <JRE_PATH>C:\Program Files\Java\jre6\</JRE_PATH> to your JRE6 HOME directory
    c. <PATH>C:\Maveryx\demo\AUT\examples.jar;</PATH>[7] to the full path of the *examples.jar* file in MAVERYX_HOME/demo/AUT
6. Run the test script as JUnit Test

[5] This is not necessary if you have installed Maveryx on Windows using the setup.

[6] See note **Error! Bookmark not defined.**.

[7] On UNIX-like and MAC OS X systems change '\' to '/' and ';' to ':'

(e.g. <PATH>/Maveryx/demo/AUT/examples.jar:</PATH>)

7. Check the log file in MAVERYX_HOME/bin/log

4.1.3. Running the Java test class example

1. Create a new project as in 4.1.1
2. Select the project's folder **src**
3. Open the resource *org.maveryx.demo.jclass* → *MaveryxTestClass.java*
4. Change[8] the value of the string *pathname* to the full path to the *PasswordDemo.xml* file (in MAVERYX_HOME/demo/AUT)
5. Open file PasswordDemo.xml and change:
    a. <WORKING_DIR>C:\Maveryx\demo\AUT</WORKING_DIR> to the MAVERYX_HOME/demo/AUT directory
    b. <JRE_PATH>C:\Program Files\Java\jre6\</JRE_PATH> to your JRE6 HOME directory
    c. <PATH>C:\Maveryx\demo\AUT\examples.jar;</PATH>[9] to the full path of the *examples.jar* file in MAVERYX_HOME/demo/AUT
6. Run the test script as Java Application
7. Check the log file in MAVERYX_HOME/bin/log

4.1.4. Running the JUnit Test examples

1. Create a new project as in 4.1.1
2. Select the project's folder **src**
3. Open the resource *org.maveryx.demo.junit* → *ButtonTest.java*
4. Change[10] the value of the string *pathname* to the full path to the *ButtonDemo.xml* file (in MAVERYX_HOME/demo/AUT)
5. Open file ButtonDemo.xml and change:
    d. <WORKING_DIR>C:\Maveryx\demo\AUT</WORKING_DIR> to the MAVERYX_HOME/demo/AUT directory
    e. <JRE_PATH>C:\Program Files\Java\jre6\</JRE_PATH> to your JRE6 HOME directory
    f. <PATH>C:\Maveryx\demo\AUT\examples.jar;</PATH>[11] to the full path of the *examples.jar* file in MAVERYX_HOME/demo/AUT
6. Run the test script as JUnit Test
7. Check the log file in MAVERYX_HOME/bin/log

---

[8] See note **Error! Bookmark not defined.**.

[9] See note 7.

[10] See note **Error! Bookmark not defined.**.

[11] See note 7.

The same applies for all other compilation unit under *org.maveryx.demo.junit*.

4.2 Writing an automated test

This section provides the source code and instructions for creating a simple automated test using Maveryx. General information about the process of creating tests can be located in Chapter 5.

4.2.1. Creating a new Test Script using JUnit

1. Create a new Maveryx Test Project as in 3.3
    a. enter a *Project name* such as "MyTestProject"
2. Create a new Maveryx Test Class as in 3.4
    a. enter a name for the *Package* such as "maveryx.test"
    b. enter a *Name* for the test case such as "MyTest"

In the following code, a simple JUnit Test is created to test the *Password Demo* application in the Maveryx distribution. This code demonstrates the basic steps that every automated test using Maveryx should take.

The demo application[12] provides a window with a text field for password entry. Users must enter the right password for authenticating. The password is "bugaboo". For security reasons, the field displays a character different from the one typed, such as '•••••'. The application checks the value typed in by the user and displays the appropriate message (success or failure).

In the following test case you will:

1. launch the application under test
2. check the password field is editable
3. enter the right password ("bugaboo")
4. check the "OK" button is enabled
5. press the "OK" button to confirm the password
6. check the successful authentication message
7. close the application under test

First of all, set to the full path to the *PasswordDemo.xml* file (in MAVERYX_HOME/demo)

```
private final String filepath = "C:\\Maveryx\\demo\\AUT\\PasswordDemo.xml";
```

The following code in the setUp() method starts the application under test (**Figure 16**).
The static method *startApplication*(*filepath*) in class Bootstrap launches the application.

```
@Before
public void setUp() throws Exception {
    //start the application under test
    Bootstrap.startApplication(filepath);
}
```

[12] http://java.sun.com/docs/books/tutorial/uiswing/components/passwordfield.html

Figure 16 – The Application Under Test

The tearDown() code closes the application under test.

```
@After
public void tearDown() throws Exception {
    //close the application under test
    GuiFrame frame = new GuiFrame();
    frame.close();

    //close the JVM running the application under test
    Bootstrap.stop(pathname);
}
```

The code in setPassword() method allows testing the application.

```
@Test
public void setPassword() throws Exception {

    GuiPasswordText t = new GuiPasswordText();
    assertTrue(t.isEditable());   //check whether the text field is editable
    t.setText("bugaboo"); //enter the password

    GuiButton ok = new GuiButton("OK");
    assertTrue(ok.isEnabled()); //check whether the push button is enabled

    //click the 'OK' button in the main frame to confirm the entered password
    ok.click();


    GuiDialog dialog = new GuiDialog();

    GuiLabel message = new GuiLabel("Success!", dialog);

    //check whether the message dialog contains the expected user message
    String expectedMessage = "Success! You typed the right password.";
    assertEquals(message.getActualId(), expectedMessage);

    //close the message dialog
    dialog.close();

    //Alternatively,  close the message dialog by clicking the OK button
    //ok.setContainer(dialog);
    //ok.click();
}
```

The following instructions enter a text (i.e. a password) into the PasswordText field (**Figure 17**):

```
GuiPasswordText t = new GuiPasswordText();
t.setText("bugaboo");
```

This line checks whether the PasswordText field is editable:

```
assertTrue(t.isEditable());
```

Figure 17 – Enter the password in the text field

The next instructions allow clicking the "OK" button in the main frame:

```
GuiButton ok = new GuiButton("OK");
ok.click();
```

This code checks whether the push button is enabled:

```
assertTrue(ok.isEnabled());
```

The following lines allow creating a new label instance:

```
GuiDialog dialog = new GuiDialog();
GuiLabel message = new GuiLabel("Success!", dialog);
```

message is defined as a new label (GuiLabel) containing the text "Success!" and comprised in a message dialog (dialog) – **Figure 18**.

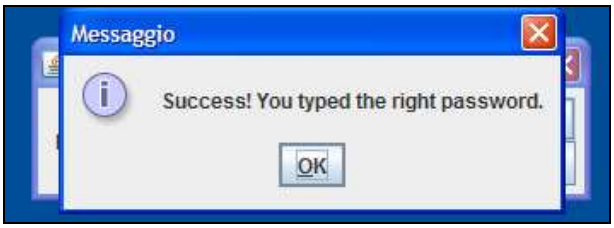


Figure 18 – The Message Dialog

The code below verifies the text of the message against the expected one:

```
String expectedMessage = "Success! You typed the right password.";
assertEquals(message.getActualId(), expectedMessage);
```

The instruction message.getActualId() retrieves the current label's text.

This line closes the message dialog:

```
dialog.close();
```

Such dialog is closed by clicking on the close button on the titlebar.

Alternatively you can close the message dialog by clicking on the 'OK' button:

```
ok.setContainer(dialog);
ok.click();
```

The first instruction ok.setContainer(dialog) allows locating the 'OK' button contained in the message dialog (dialog). This is necessary to avoid ambiguities, because there is another 'OK' button in the main frame (**Figure 19**).

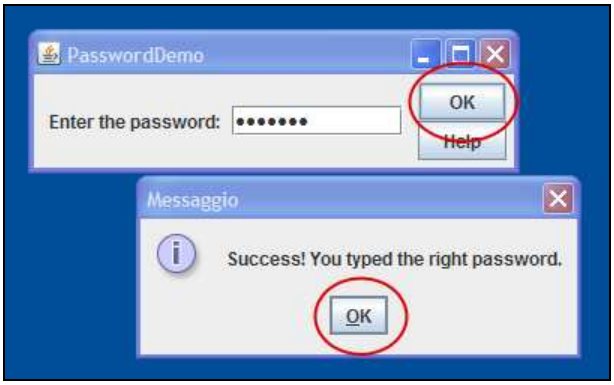


Figure 19 – Multiple 'OK' buttons

***Note*:** see the Javadoc documentation for more details on the used (and all other) APIs.

5. Testing with Maveryx

This chapter describes how to write tests that use the Maveryx framework resources and to set special properties required by the tests.

This chapter explains how to:

- create a new test project (5.1)
- configure the launch file to execute the application under test (5.2)
- configure the GUI Object Finder engine for optimal performance (5.3)
- create automated tests using the provided API (5.4)

5.1 Creating a new test project

You can use Maveryx with any Java editor or IDE you want. To create a new test project using Maveryx you have to add the following libraries (JAR files) to the build path settings of your Java test project.

1. ***bootstrap.jar*** in MAVERYX_HOME/bin
2. ***cajo.jar*** in MAVERYX_HOME/lib
3. ***proxy.jar*** in MAVERYX_HOME/bin
4. ***factory.jar*** in MAVERYX_HOME/bin
5. ***finderXMLObject.jar*** in MAVERYX_HOME/bin
6. ***jdom.jar*** in MAVERYX_HOME/lib
7. ***utils.jar*** in MAVERYX_HOME/bin
8. ***junit4runner.jar*** in MAVERYX_HOME/lib
9. ***testReport.jar*** in MAVERYX_HOME/bin
10. ***freemarker.jar*** in MAVERYX_HOME/lib
11. ***jcommon-1.0.16.jar*** in MAVERYX_HOME/lib
12. ***junit.jar*** in MAVERYX_HOME/lib
13. ***jaxen.jar*** in MAVERYX_HOME/lib
14. ***jxl.jar*** in MAVERYX_HOME/lib

NOTE: If you are using the Maveryx Eclipse plug-in this operation is performed automatically when creating a new test project.

If you are using Eclipse without the Maveryx plug-in:

1. Right click the *project name/node* and navigate to **Build Path**
2. Select **Configure Build Path...** and the project properties window will appear showing your build path configurations
3. Select the **Libraries** tab
4. Click **Add External JARs...**
5. *Locate and select the JARs and click OK*. The JARs will now appear in the list of libraries in the build path
6. Click **OK** to close the properties window.

If you are using NetBeans:

1. Right click the *project name/node* and select **Properties**
2. select the **Libraries** category
3. Click **Add JAR/Folder**
4. *Locate and select the JARs and click OK*. The JARs will now appear in the list of libraries in the build path
5. Click **OK** to close the properties window.

5.2 Configuring the launch file

To execute the application under test you have to create the related launch file. This is an XML file that contains the application launch settings (e.g. main class, arguments, classpath, JRE …).

There are two methods of creating launch files:

- You can use the ***Application Configuration*** tool[13] to quickly create and edit a launch file
- You can manually create a launch file

5.2.1. Using the Application Configuration tool

The *Application Configuration* tool is an intuitive, GUI-based application which enables you to easily create, save and modify launch files.

To start the Application Configuration tool on Windows, select ***Start*** → ***Programs*** → ***Maveryx*** → ***Application Configuration***

From Eclipse, click the [ ] button on the toolbar.

The Application Configuration main window (**Figure 20**) consists of the following properties:

- ***Application Name***: the application name
- ***Application Description***: a short application description
- ***Main Class***: the fully qualified name of the main class for the application to be tested
- ***Working Directory***: the working directory that will be used during application execution
- ***Program Arguments***: the arguments to pass to the main class during application execution.
- ***VM Arguments***: the Java Virtual Machine options[14] that should be used for running the main class
- ***Runtime JRE***: the target Java Runtime Environment to run the application under test
- ***Classpath***: the classpath for running the application. Use the buttons on the right side of the panel to add and remove the classpath elements. You can add any of the following:
    - *JAR file*: a Java archive file somewhere on your system
    - *Class folder:* a folder, somewhere on your system, with a collection of class files

[13] The Application Configuration tool is available for Subscribers only.

[14] For more information about VM options, see http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

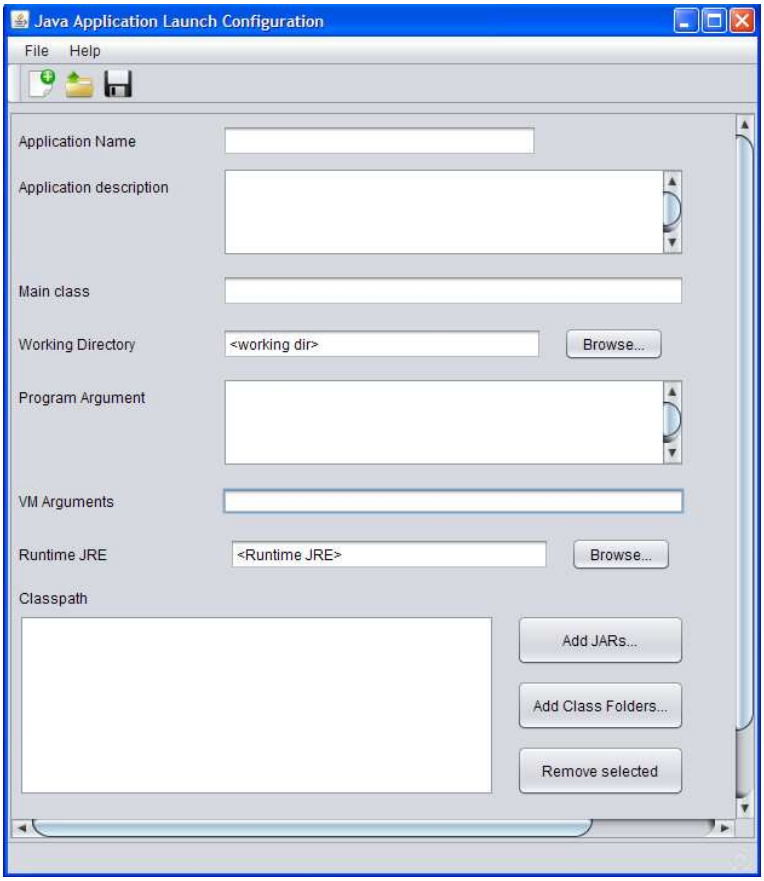


Figure 20 – The Application Configuration tool

5.2.2. Setting up the launch file

This section describes the XML format used for launch files.

To execute the application under test you have to create an XML launch file as the one below[15] (available in MAVERYX_HOME/demo/AUT).

```
<?xml version="1.0" encoding="UTF-8"?>
<AUT_DATA>

    <WORKING_DIR>C:\Maveryx\demo</WORKING_DIR>

    <APPLICATION_NAME>PasswordDemo</APPLICATION_NAME>

    <AUT_ARGUMENTS></AUT_ARGUMENTS>

    <VM_ARGUMENTS></VM_ARGUMENTS>

    <DESCRIPTION>PasswordDemo test</DESCRIPTION>

    <JRE_PATH>C:\Program Files\Java\jre6\</JRE_PATH>

    <MAIN_CLASS>com.sun.demo.PasswordDemo</MAIN_CLASS>
```

[15] We recommend you start from the provided example.

```
    <!-- on UNIX-like and MAC OS X systems change the path separator ';' to ':' -->
    <CLASSPATH>
        <LIB>
            <PATH>C:\Maveryx\demo\example.jar;</PATH>
        </LIB>
    </CLASSPATH>
</ AUT_DATA>
```

AUT_DATA: the root element.

WORKING_DIR: the directory from which the application under test is launched.

APPLICATION_NAME: the name of the application to test.

AUT_ARGUMENTS: the arguments to pass to the application's *main* method.

VM_ARGUMENTS: the Java VM arguments to use to run the application under test. Refer to java help for more details.

DESCRIPTION: a textual description of the application to test. [optional]

JRE_PATH: the full file system path to the HOME directory containing the Java executable used to run the application under test.

MAIN_CLASS: the name of the main class to be invoked.

CLASSPATH: this element contains the classpath. CLASSPATH shall consist of one *or more LIB* elements.

LIB: specifies a JAR file needed to execute the application under test. An element LIB shall consist of *PATH*.

```
<!-- on UNIX-like and MAC OS X systems change the path separator ';' to ':' -->
<LIB>
   <PATH>C:\Maveryx\demo\example.jar;</PATH>
</LIB>
```

PATH: the full file system path to a JAR file needed to execute the application under test

If your application classpath consists of *n* JAR files, you can create *n* LIB elements, one for each JAR, or only one LIB element concatenating all paths.

For example, if your classpath consists of *lib1.jar*, *lib2.jar* and *lib3.jar* in *C:\myApp* you can create:

```
<!-- on UNIX-like and MAC OS X systems change the path separator ';' to ':' -->
<LIB>
   <PATH>C:\myApp\lib1.jar;</PATH>
</LIB>
<LIB>
   <PATH>C:\myApp\lib2.jar;</PATH>
</LIB>
<LIB>
   <PATH>C:\myApp\lib3.jar;</PATH>
</LIB>
```

or

```
<!-- on UNIX-like and MAC OS X systems change the path separator ';' to ':' -->
<LIB>
   <PATH>C:\myApp\lib1.jar;C:\myApp\lib2.jar;C:\myApp\lib3.jar;</PATH>
</LIB>
```

5.3 Configuring the GUI Object Finder

Before using Maveryx you can configure the *GUI Object Finder* through the ***SearchingStrategyPlugInSettings.xml*** file in MAVERYX_HOME/bin.

This xml file allows setting:

- the *searching criteria* to be used to find the GUI objects to test
- the parameters of the *evaluation function* to estimate the goodness of the test object (best candidate) found and to disambiguate in case of multiple objects

You can add your own searching criteria and evaluation function, which best fit your application's domain, by implementing the relevant interfaces (see the Javadoc documentation).

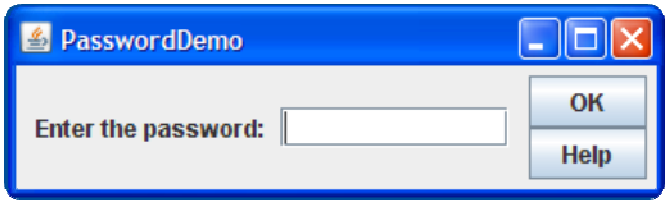


Figure 21 – The demo application

5.3.1. Configure the Searching Criteria

Maveryx comes by default with three searching criteria:

- NameRole
- AssociatedTooltip
- AssociatedLabel

NameRole allows locating a GUI object to test by its *name* a/o *role*.
The *name* is the object's label or caption, the *role* is the function the object plays in the UI.

For example:

```
//By NAME and ROLE
GuiFrame f = new GuiFrame("PasswordDemo"); // ROLE = frame; NAME = 'PasswordDemo'

//Only by ROLE
GuiFrame f = new GuiFrame(); //ROLE = frame
```

AssociatedTooltip allows locating a GUI object to test by its *tooltip*.
Sometimes GUI controls don't have a label or caption and can be identified by their tooltips.
This is the case of buttons in a toolbar (**Figure 22**).

Figure 22 – Tooltip on toolbar button

```
GuiButton save = new GuiButton("Save"); //create a button specifying its tooltip
save.click(); //click the 'Save' button
```

AssociatedLabel allows locating a GUI object to test by its *label*.
This is the case of Text Fields that are identified by their associated label (**Figure 23**).

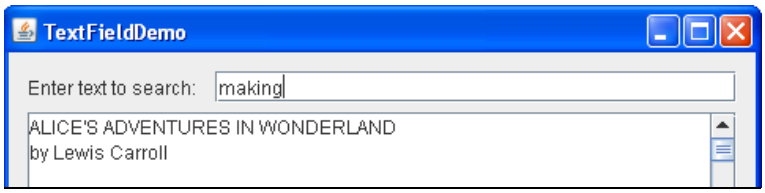


Figure 23 – Associated Label on text field

```
//define the text field by its associated string label
GuiText t = new GuiText("Enter text to search");
t.setText("making"); //enter the text into the text field
```

Each criterion has two main properties *WEIGHT* and *IS_ACTIVE*.

```
<CRITERION NAME="NameRole"
 CLASS="plugin.searchingStrategy.javaAccessibilitySnapshotFinder.SearchingCriterionNameRole"
 WEIGHT="1"
 IS_ACTIVE="true">
```

IS_ACTIVE = *true* means that the criterion is enabled, *false* otherwise.

WEIGHT = float number that specify the criterion importance. This value is used by the GUI Object Finder to calculate the score of each candidate (**Figure 24**).

```
************************************
** Result table **
- Result: id='a2' score: 1.0
    +Criterion: NameRole
        +Mode: ExactMatch
    +Criterion: AssociatedTooltip
        +Mode: ExactMatch
    +Criterion: AssociatedLabel
        +Mode: ExactMatch
************************************
```

Figure 24 – Example of score

Each criterion has a set of ***rules***.
Maveryx, by default, uses the following rules:

- ExactMatch
- CaseInsensitiveMatch
- PartialMatch
- Wildcard
- LevenshteinDistance
- SynonymousMatch
- GoogleTranslation

ExactMatch: allows searching for the GUI object to test as it is 'exactly' defined in the test script.

For example:

```
//the given frame id matches exactly the real frame title
GuiFrame f = new GuiFrame("PasswordDemo");
f.close();
```

CaseInsensitiveMatch: same as ExactMatch, but case–insensitive.

For example:

```
//the given frame id matches the real frame title but lower case
GuiFrame f = new GuiFrame("passworddemo");
f.close();
```

PartialMatch: allows locating the test object by matching partial data (substring matching).

For example:

```
//the given frame id partially matches (substring) the real frame title
GuiFrame f = new GuiFrame("Demo");
f.close();
```

Wildcard: allows finding the GUI object to test by using the Java regular expressions (refer to http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html for the regular–expression constructs).

For example:

```
//the given frame id partially matches (regular expression) the real frame title
GuiFrame f = new GuiFrame("P*Demo");
f.close();
```

LevenshteinDistance: allows searching for the test object by using the Levenshtein distance (refer to http://en.wikipedia.org/wiki/Levenshtein_distance for more information on this edit distance).

For example:

```
//the given frame id partially matches (edit distance) the real frame title
GuiFrame f = new GuiFrame("PwdDemo");
f.close();
```

SynonymousMatch: allows searching for the GUI control to test by using broader or narrower or related terms or concepts in the installed Wordnet vocabulary.

For example:

```
//the given button label is a synonymous of the real button caption
GuiButton ok = new GuiButton("O.K.");
ok.click();
```

GoogleTranslation: allows locating the test object by using translation terms.

For example:

```
//the given button label is the italian translation of the real button caption
GuiButton help = new GuiButton("Aiuto");
help.click();
```

NOTES

The following rules:

- PartialMatch
- Wildcard
- LevenshteinDistance
- SynonymousMatch

are useful to locate the GUI objects to test in case of partial information given in the test script (typical in deriving tests from vague specifications) or if such controls have changed since test creation.

The GoogleTranslation rule is useful to locate “low level” components (such as default dialog messages) that use the default input language of the operating system or in case of internationalization.

Each criterion has two main properties *FACTOR* and *IS_ACTIVE*.

```
<ALLOWED_VALUES VALUE="ExactMatch" FACTOR="1" IS_ACTIVE="true"/>
```

IS_ACTIVE = *true* means that the rule is enabled, *false* otherwise.

FACTOR = float number that specify the rule weight. This value is used by the GUI Object Finder to calculate the score of each candidate.

5.3.1.1 Configuring the LevenshteinDistance

To configure the LevenshteinDistance you have to set the value of the *maximumLevenshteinDistance* property.

```
<PROPERTY NAME="maximumLevenshteinDistance" VALUE="2" IS_ACTIVE="true"/>
```

The *maximumLevenshteinDistance* is the maximum 'edit distance'[16] between the GUI object defined in the test scripts and the real one.

If a candidate test object has a LevenshteinDistance greater that maximumLevenshteinDistance it will be discarded.

5.3.1.2 Configuring the SynonymousMatch

To configure the SynonymousMatch you have to set the value of the *wordnetdataPath* property.

```
<PROPERTY NAME="wordnetdataPath" VALUE="C:\Maveryx\lib\wordnetdata" />
```

The *wordnetdataPath* is the full file system path to the wordnetdata directory containing the WordNet dictionary.

5.3.1.3 Configuring the GoogleTranslation

To configure the GoogleTranslation you have to set:

1. the language of the application under test (AUTLanguage)
2. the Internet connection (Connection)

To set the language of the application under test you have to define the value of the *AUTLanguage* property by choosing a language from the allowed ones (English, French, German...).

```
<PROPERTY NAME="AUTLanguage" VALUE="ENGLISH">
   <ALLOWED_VALUES VALUE="ENGLISH" IS_ACTIVE="true"/>
   <ALLOWED_VALUES VALUE="FRENCH" IS_ACTIVE="true"/>
   <ALLOWED_VALUES VALUE="GERMAN" IS_ACTIVE="true"/>
   <ALLOWED_VALUES VALUE="PORTUGUESE" IS_ACTIVE="true"/>
   <ALLOWED_VALUES VALUE="SPANISH" IS_ACTIVE="true"/>
   <ALLOWED_VALUES VALUE="ITALIAN" IS_ACTIVE="true"/>
</PROPERTY>
```

To set the internet connection, in case of proxy with no authentication, you have to define the following property values:

[16] http://en.wikipedia.org/wiki/Edit_distance

- ProxyAddress : the host name of the proxy server
- Port : the port number
- Username : the username for authentication
- Password : the password for authentication

```
<PROPERTY NAME="Connection" VALUE="" IS_ACTIVE="true">
   <PROPERTY NAME="ProxyAddress" VALUE="proxy.maveryx.com" />
   <PROPERTY NAME="Port" VALUE="80" />
   <PROPERTY NAME="Username" VALUE="maveryx" />
   <PROPERTY NAME="Password" VALUE="maveryx01" />
</PROPERTY>
```

5.3.2. Configuring the Evaluation Function

To configure the ***evaluation function*** you have to set the *threshold* and the *minimumObjectResolution*.

```
<EVALUATION_FUNCTION
CLASS="plugin.searchingStrategy.javaAccessibilitySnapshotFinder.EvaluationFunction">
   <PROPERTY NAME="threshold" VALUE="0" IS_ACTIVE="true"/>
   <PROPERTY NAME="minimumObjectResolution" VALUE="0" IS_ACTIVE="true"/>
   <PROPERTY NAME="ObjectIsShowing" VALUE="" IS_ACTIVE="true"/>
   <PROPERTY NAME="ObjectIsInDialog" VALUE="" WEIGHT="1.5" IS_ACTIVE="true"/>
   </PROPERTY>
</EVALUATION_FUNCTION>
```

What is the *threshold*?
Each candidate test object retrieved by the GUI Object Finder, applying the given criteria and rules, has a *score*.
The score is calculated by summing (weighted sum) *FACTOR* and *WEIGHT* of the successful rules and criteria.
GUI objects with a score lesser then the threshold are discarded. ‘0’ means no threshold applied (i.e. all candidates are suitable).
The best candidate, the one with the higher score (greater than threshold), is returned as the GUI object to test.
If no object is found or no candidate has a score greater than threshold an *ObjectNotFoundException* is generated by the GUI Object Finder.

The *minimumObjectResolution* is the minimum score distance between two best candidates. ‘0’ means no minimum resolution applied.
If, for example, two candidates are found:

- $Candidate_1$ with score S_1
- $Candidate_2$ with score S_2
- and $S_1 > S_2$

if $S_1 - S_2 >$ *minimumObjectResolution* ⇨ $Candidate_1$ is returned as the GUI object to test. Otherwise a *MultipleObjectFoundException* is generated by the GUI Object Finder.

ObjectIsShowing allows distinguishing between showed and not showed GUI objects.

ObjectIsInDialog allows assigning higher score to GUI objects in modal dialogs.

5.4 Tricks and Tips

Like any other language or platform, Maveryx has its tricks and tips. This section contains tips, techniques, and sample code on various topics of interest using the Maveryx platform.

5.4.1. Launching the application under test

The static method *startApplication*(*String path*) in class Bootstrap launches the application.

```
Bootstrap.startApplication("C:\\Maveryx\\demo\\AUT\\PasswordDemo.xml");
```

The argument of such methods is the full path to the XML file containing the launch parameters. To configure the launch file refer to 5.2.

When you close the application under test you can use the static method *stop*(*String path*) in class Bootstrap. This method closes the JVM running the application under test.

```
Bootstrap.Stop("C:\\Maveryx\\demo\\AUT\\PasswordDemo.xml");
```

5.4.2. Creating test objects

To test a GUI object you have to create a new instance of such control.

For example, to click the 'OK' in **Figure 21** you have to create first an instance of this button.

You can define the ok button (as a new instance of the class GuiButton) by giving its identifier[17] (the label "OK").

```
GuiButton ok = new GuiButton("OK");
ok.click();
```

Or, you can define the ok button without passing any argument to the constructor.

```
GuiButton ok = new GuiButton();
ok.click();
```

This is a very fast way to define a button, but if there is another button, with any label, the system returns a *MultipleObjectFoundException* (as for the application in **Figure 21** that has two push buttons "OK" and "Help").

Or, you can define the ok button specifying its label ("OK") and its container object (GuiFrame f).

```
GuiFrame f = new GuiFrame("PasswordDemo");
GuiButton ok = new GuiButton("OK",f);
ok.click();
```

[17] The *identifier* of a GUI object could be its label/caption or its tooltip.

This is most complete way to define a push button and this is useful when you have a button with the same label in different windows such as in **Figure 19**.

You can apply the above concepts to all GUI objects supported by Maveryx.

5.4.2.1 Creating multiple objects

If you have to test multiple GUI object with the same role you can:

- create multiple instances, one for each object or
- create one instance and use the ok.setId(*String*) method to refer to each object

For example, to click the 'OK' button and then the "Help" one in **Figure 21** you can

Create multiple button instances, one for each button to test:

```
GuiButton ok = new GuiButton("OK");
ok.click();
…
…
GuiButton help = new GuiButton("Help");
help.click();
```

Or, you can create one 'generic' button instance, and use the ok.setId(*String*) method to refer to each one:

```
GuiButton b = new GuiButton();
b.setId("OK")
b.click();
…
…
b.setId("Help")
b.click();
```

The method setContainer(*String*) apply the same to the container object.

For the example in **Figure 19** you can:

```
GuiFrame f = new GuiFrame("PasswordDemo");
GuiButton ok = new GuiButton("OK",f);
ok.click();
…
…
GuiDialog dialog = new GuiDialog();
ok.setContainer(dialog);
ok.click();
```

The instruction ok.setContainer(dialog) allows locating the 'OK' button in the message dialog (dialog).

You can apply the above concepts to all GUI objects supported by Maveryx.

5.4.3. Getting info on test objects

When you create a test object you can specify its identifier (label or tootip). Algorithm in 5.3.1 allows creating GUI objects giving partial or approximate identifier.
You can get the actual identifier of the GUI object by using the getActualId() method.

For the application in **Figure 21**, you can check the title of the frame by these instructions:

```
GuiFrame f = new GuiFrame();
assertEquals(f.getActualId(),"PasswordDemo");
```

Or, if you give a partial identifier you can re–set the object id with the actual one:

```
GuiFrame f = new GuiFrame("Demo");
f.setId(getActualId());
```

This instruction changes the GUI object identifier from Demo to PasswordDemo.

The same apply for the GUI object role by using the getActualRole() method.

Note: when you use the methods getId and getRole you deal with identifier and role of the Java object, not the actual ones.

```
GuiFrame f = new GuiFrame("Demo");
System.out.println(f.getId());
System.out.println(f.getActualId());
```

The first print instruction prints Demo while the second one PasswordDemo.

5.4.4. waitForObject()

The waitForObject() method waits until the GUI object is available.
For example, if you have to wait for a message dialog after downloading a file you can use the waitForObject(int timeOut,int step) method.
The timeOut specify when stop waiting, the step specify the time interval between successive attempts. The framework will search for the GUI object to test each time step until the object is found or the timeOut has reached.
For example:

```
GuiDialog dialog = new GuiDialog();
GuiButton ok = new GuiButton("OK",f);
ok.waitForObject(20, 1).click();
```

Note: by default, if not specified, the framework waits 15 seconds for the GUI object to test retrying each second.

If the timeOut has expired an *ObjectNotFoundException* is generated.

5.4.5. Logging

Each time a test case is executed Maveryx records log information into a file.
For each executed test, Maveryx creates a log folder into the MAVERYX_HOME/bin/log directory.
The name of the log folder is **log_*date_time*** (e.g. log_2011-10-26_16.13.41).
Each log folder contains:

- **log_*date_time*.txt** file (e.g. log_2011-10-20_14.49.39.txt)
- [optional] one or more **snapshot-*date_time*.snp** (e.g. snapshot-2011-10-26_16.14.30.snp)
- [optional] one or more **screenCapture-*date_time*.jpg** (e.g. screenCapture-2011-10-26_16.14.30.jpg)

The .txt file records valuable information on:

- configuration settings (both user–defined and platform–specific ones)
- GUI Object Finder activities
- robot activities
- errors and exceptions occurring (for tracing various different potential problems)

The .snp and .jpg files are created in case of a GUI Object Finder failure, and contain a snapshot of the current user interface.

5.4.6. Reporting

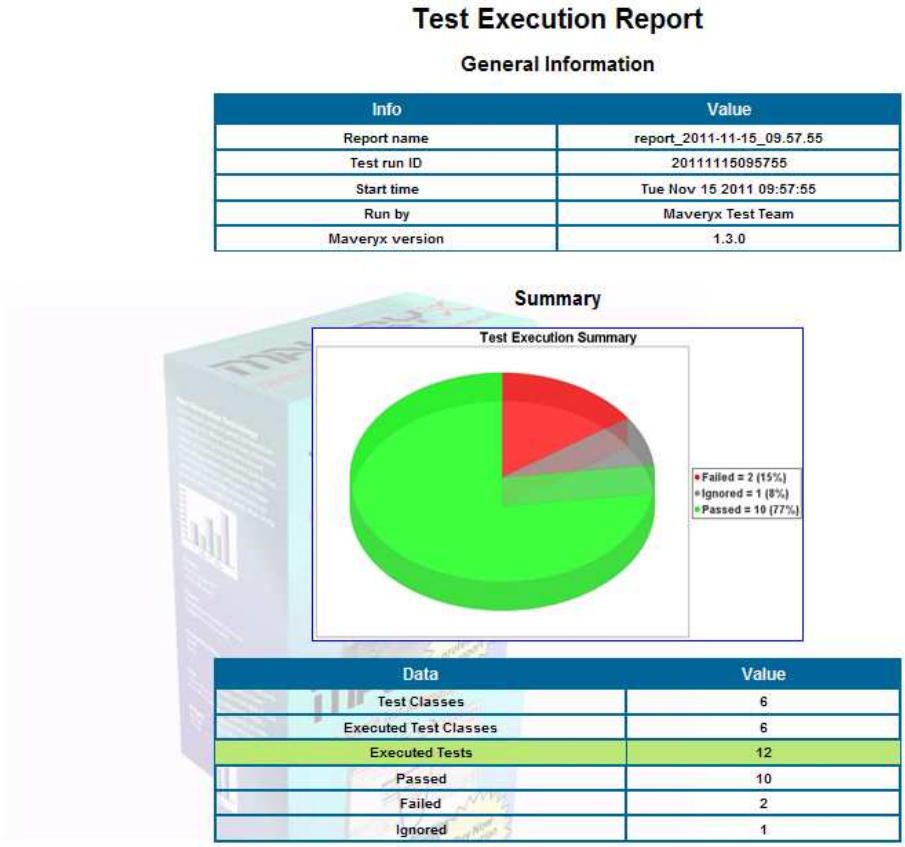
After the tests complete, Maveryx creates a report of the test run (**Figure 25**). The report includes a summary that lists the tests and their execution result status (passed, failed, and ignored).
For each test run, Maveryx creates a report folder into the MAVERYX_HOME/bin/report directory.
The name of the report folder is **report_*date_time*** (e.g. report_2011-11-04_09.36.22).
Each report folder contains the following files:

- **testReport.xml**: the test report containing all test results
- **summaryChart.png**: this chart displays the number of passed, ignored, or failed tests
- **classesChart.png**: this chart displays, for each test class, the number of passed, ignored, or failed tests

File testReport.xml includes:

- Information on the test run
- A test summary, including the number of executed, passed and failed tests
- Pass/Fail results of each test

You can use Internet Explorer or Mozilla Firefox to view testReport.xml just as you view any HTML page.



Test Execution Report

General Information

Info	Value
Report name	report_2011-11-15_09.57.55
Test run ID	20111115095755
Start time	Tue Nov 15 2011 09:57:55
Run by	Maveryx Test Team
Maveryx version	1.3.0

Summary

Data	Value
Test Classes	6
Executed Test Classes	6
Executed Tests	12
Passed	10
Failed	2
Ignored	1

Figure 25 – Maveryx Test Report

5.4.7. Data–driven testing

Data–driven tests are functional tests that perform the same series of test actions, but with different data sets supplied by an external data source (database, spreadsheet, XML files …). Maveryx supports XML files, Excel spreadsheets and just about any database (JDBC sources) including Microsoft SQL Server, Oracle, MySQL, Microsoft Access, and others for data–driven testing.

In the MAVERYX_HOME/demo directory there are some examples using the data–driven approach.

5.4.7.1 Getting test data from an XML file

Maveryx supports XML data sources with the following structure:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
   <testdata name="username" type="java.lang.String">
    <data id="user1" value="john.smith@maveryx.com" />
    <data id="user2" value="support@maveryx.com" />
  </testdata>
  <testdata name="password" type="java.lang.String">
    <data id="1" value="Jsmith01" />
    <data id="2" value="johnsmith" />
    <data id="3" value="mysupport" />
  </testdata>
</dataset>
```

dataset: the root element. This element shall contain at least of one *testdata* element.

testdata: contains a collection of related test data. Such element shall contain at least of one *data* element. *testdata* shall consist of the following properties:

- *name*: a string label for the test data
- *type*: the test data type (OPTIONAL)

data: the test data. This element shall consist of the following properties:

- *id*: a string label specifying the data IDentifier (you can use alphanumeric values such as "1", "one", "data–1"…)
- *value*: the test data value

To integrate data–driven testing into your Maveryx test script you have to first import the relevant libraries:

```
import org.maveryx.util.datadriven.*; //import the data–driven libraries
```

Then you have to create a new TestDataManager object to manage the test data, by passing the full path to the XML data file:

```
TestDataManager t = new TestDataManager("C:\\Maveryx\\demo\\data\\Test_Data.xml")
```

To get a specific value you have to specify the *testdata* element and the specific *data*:

```
//get the 'username' with id = 'user1': → 'john.smith@maveryx.com'
String username = t.getData("username").getDataById("user1").getValue();

//get the 'password' with id = 1: → 'Jsmith01'
String password = t.getData("password").getDataById("1").getValue();
```

This piece of code shows how to integrate the data–driven approach into a Maveryx test script:

```
//create a new password text field
GuiPasswordText pText = new GuiPasswordText();

//get the 'password' with id = 2
String password = t.getData("password").getDataById("2").getValue();

//enter the password
```

```
pText.setText(password);
```

5.4.7.2 Getting test data from an Excel file

Maveryx supports Excel data sources with the structure in **Figure 26**.

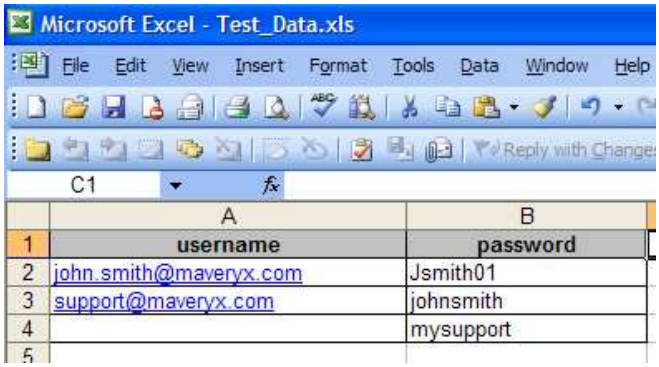


Figure 26 – Excel data file

Test data are organized by columns. The first row contains the table header. The first cell of each column shall contain the string label for the test data (e.g. "username", "password"). Each cell from the second one defines a test data (input or expected output).

To integrate data–driven testing into your Maveryx test script you have to first import the relevant libraries:

```
import org.maveryx.util.datadriven.*; //import the data–driven libraries
```

Then you have to create a new TestDataManager object to manage the test data, by passing the full path to the xls data file:

```
TestDataManager t = new TestDataManager("C:\\Maveryx\\demo\\data\\Test_Data.xls")
```

NOTE: the TestDataManager constructor creates automatically an equivalent XML file (5.4.7.1) from the XLS one.

To get a test data you have to specify the *column header* and the *row number*:

```
//get the 'username' at row = 2 → 'john.smith@maveryx.com'
String username = t.getData("username").getDataById("2").getValue();

//get the 'password' at row = 3 → 'johnsmith'
String password = t.getData("password").getDataById("3").getValue();
```

This piece of code shows how to integrate the data–driven approach into a Maveryx test script:

```
//create a new password text field
GuiPasswordText pText = new GuiPasswordText();

//get the 'password' at row = 2 → 'Jsmith01'
String password = t.getData("password").getDataById("2").getValue();
```

```
//enter the password
pText.setText(password);
```

5.4.8. Verification Points

Verification points verify that an application under test function as expected. They are comparison operations useful for identifying potential defects.
You can add verification points to check various aspects of an application–under–test, such as: the properties of an object, text within a text area, data within a table, etc.

Maveryx allows you to use assertions to verify that the actual results match the expected results. Each assertion contains a boolean expression that you believe will be true when the assertion executes. You can use JUnit assertions to test your assumptions about the application–under–test.

The most common type of verifications point is to compare the actual value of an object's property with an expected value.
For example, you can check whether a button is enabled or disabled:

```
GuiButton b = new GuiButton("OK"); //constructs a new push button
assertTrue(b.isEnabled());//check the button is enabled
```

Or, after a text was entered, you can verify that the line editor's text property returns the expected value:

```
GuiText t = new GuiText(); //constructs a new text field
String text = "This is a test...."; //the text to be set
t.setText(text); //enter the text
assertEquals(t.getText(), text); //check the entered text
```

5.4.9. Distributed and Multi-Application Testing

Maveryx communicates with the application under test via a network TCP/IP connection. This allows driving applications on different machines and platforms from one central place.
In this section you will see how to remotely run tests on a different machine. For example, let's assume that you are working on a computer A, and that you want to test an application located on a computer B.
The first step is to install Maveryx and the application under test on the target computer (computer B). Once you have created the launch file (5.2) for the application under test, put it in the MAVERYX_HOME/bin/remote directory on computer B.
Start the Maveryx server on computer B by executing:

- The Remote Server tool[18] (**Figure 27**) or
- The batch file MAVERYX_HOME/bin/Remote_Server_START.bat[19]

[18] The Remote Server tool is available for Subscribers only.

[19] On UNIX, Linux and Mac OS systems you have to 'translate' the batch file to a shell script and launch it.

To start the Remote Server tool on Windows, select ***Start*** → ***Programs*** → ***Maveryx*** → ***Remote Server*** or the

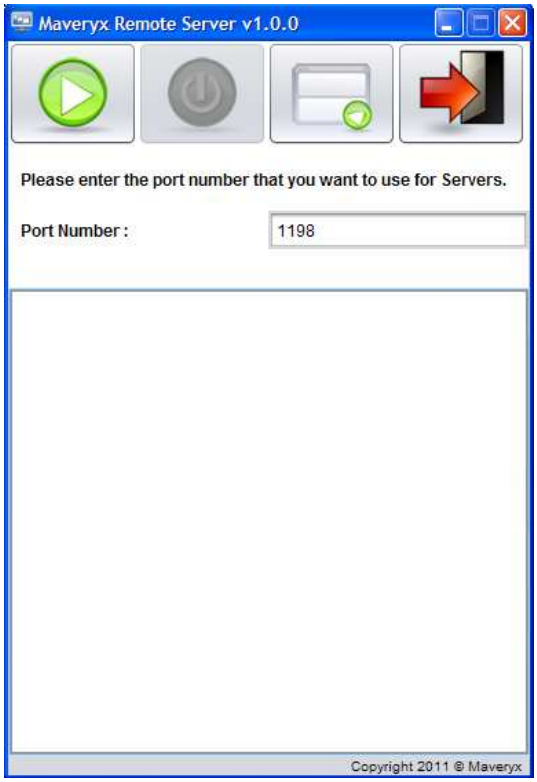

Maveryx Remote Server v1.0.0
Please enter the port number that you want to use for Servers.
Port Number :
1198
Copyright 2011 © Maveryx

Figure 27 – Maveryx Remote Server tool

On the console get the port number used by the server (e.g. "Bootstrap started on port: 1198"). By default, Maveryx server uses port 1198.

To run the test on a remote machine you have to add the following code to the constructor (in order to connect from computer A to the Maveryx server on computer B):

```
session = new SessionData();
session.setRemoteIP(ip_address); //IP address or host name of the remote computer (computer B)
session.setRemotePort(port_number); //port number used by the Maveryx server (on computer B)
```

For example:

```
public ButtonTest() throws Exception {
    super();
    session = new SessionData();
    session.setRemoteIP("168.212.226.204");
    session.setRemotePort(1198);
}
```

You are now ready to create test cases on computer A and have them executed on computer B.